Address space layout randomization next generation

Marco Gisbert, Hector; Ripoll, Ismael

Published in:
Applied Sciences

DOI:
10.3390/app9142928

Published: 22/07/2019

Document Version
Publisher's PDF, also known as Version of record

applied sciences

MDPI

Article

Address Space Layout Randomization Next Generation

Hector Marco-Gisbert [1,*] and Ismael Ripoll Ripoll [2]

[1] School of Computing, Engineering and Physical Sciences, University of the West of Scotland, High Street, Paisley PA1 2BE, UK
[2] Department of Computing Engineering, Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain
* Correspondence: hector.marco@uws.ac.uk; Tel.: +44-1418494418

Received: 2 June 2019; Accepted: 15 July 2019; Published: 22 July 2019

Abstract: Systems that are built using low-power computationally-weak devices, which force developers to favor performance over security; which jointly with its high connectivity, continuous and autonomous operation makes those devices specially appealing to attackers. ASLR (Address Space Layout Randomization) is one of the most effective mitigation techniques against remote code execution attacks, but when it is implemented in a practical system its effectiveness is jeopardized by multiple constraints: the size of the virtual memory space, the potential fragmentation problems, compatibility limitations, etc. As a result, most ASLR implementations (specially in 32-bits) fail to provide the necessary protection. In this paper we propose a taxonomy of all ASLR elements, which categorizes the entropy in three dimensions: (1) how, (2) when and (3) what; and includes novel forms of entropy. Based on this taxonomy we have created, ASLRA, an advanced statistical analysis tool to assess the effectiveness of any ASLR implementation. Our analysis show that all ASLR implementations suffer from several weaknesses, 32-bit systems provide a poor ASLR, and OS X has a broken ASLR in both 32- and 64-bit systems. This is jeopardizing not only servers and end users devices as smartphones but also the whole IoT ecosystem. To overcome all these issues, we present ASLR-NG, a novel ASLR that provides the maximum possible absolute entropy and removes all correlation attacks making ASLR-NG the best solution for both 32- and 64-bit systems. We implemented ASLR-NG in the Linux kernel 4.15. The comparative evaluation shows that ASLR-NG overcomes PaX, Linux and OS X implementations, providing strong protection to prevent attackers from abusing weak ASLRs.

Keywords: security; internet of things address space layout randomisation; vulnerability analysis; protection techniques

1. Introduction

Address Space Layout Randomization (ASLR) is a well-known, mature and widely used protection technique which randomizes the memory address of processes in an attempt to deter forms of exploitation [1], which rely on knowing the exact location of the process objects. Rather than increasing security by removing vulnerabilities from the system, as source code analysis tools [2] tend to do, ASLR is a prophylactic technique which tries to make it more difficult to exploit existing vulnerabilities [3].

Unlike other security methods [4,5], the security provided by ASLR is based on several factors [6], including how predictable the random memory layout of a program is, how tolerant an exploitation technique is to variations in memory layout and how many attempts an attacker can make practically.

ASLR is a wide spectrum protection technique, in the sense that rather than addressing a special type of vulnerability, as the renewSSP [7] does, it jeopardizes the programming code [8]

of the attackers independently of the vector [9] used to inject code or redirect the control flow. Similarly to other mitigation techniques, ASLR mitigates code execution attacks by crashing the application, and so the attack is degenerated into a denial of service.

The ASLR is an abstract idea which has multiple implementations [10–13], though there are important differences in performance and security coverage between them. We therefore need to make a clear distinction between the core concept of ASLR, which is typically described as something which "*introduces randomness in the address space layout of user space processes*" [14], and the exact features of each implementation.

Although ASLR is more than 14 years old [15], it is still a very effective protection against modern attacks [16,17] there is still a lot of work and innovations to be done, both on the design and the implementation. For example, the implementation of the KASLR (Kernel ASLR), which loads kernel code and drivers or modules in random positions [18] is still under development and improvement in some ecosystems such as the IoT [19] and other 32-bit low powered devices.

This paper is organized as follows: a full taxonomy of the ASLR is presented in Section 2, followed by a critical analysis of limitations of current ASLR designs in Section 3. Later, we introduce ASLRA in Section 4, a tool t automatically analyze and detect ASLR weaknesses. Section 5 presents the weaknesses found in Linux, PaX and OS X. Then in Section 6 we describe the constraints that must be taken into account when designing a practical ASLR. In Section 7 we propose a new ASLR named ASLR-NG, which overcome all limitations and weaknesses. Section 8 evaluates our proposal in a real implementation, and finally, Section 9 concludes the paper.

2. ASLR Taxonomy

In this section, we present our novel taxonomy which consider three different dimensions to assess practical ASLR implementations. This taxonomy revealed the security issues described in Section 5 and are were used to develop the assessment tool presented in Section 4.

Depending on the exact implementation details there may be important differences in the final operation and effectiveness of the ASLR, and so in order to understand and compare these differences between ASLR implementations, it is necessary to have a detailed definition of all memory objects and how they can be randomized.

In what follows, a memory object is classed as a block of virtual memory allocated to a process, examples of which include the stack, the executable, a mapped file, an anonymous map and the vDSO. For our purposes, the size and the base address of each object are the most relevant attributes.

Several objects may be allocated together (in consecutive addresses) with respect to a random address base, which will be referred to as the **area** or as the **zone**. For example, in Linux, all objects allocated via the `mmap()` system call are placed side by side in the `mmap_area` area.

We have categorized the ASLR into three main dimensions: (1) When, (2) What and (3) How. The first dimension defines how often randomization takes place, the second determines which objects are randomized and the last one defines how and how much the objects are randomized.

2.1. Dimension 1: When

The When dimension defines when the entropy used to randomize the object is generated. For example, per-exec is when the random addresses are generated when a new image is loaded in memory. Linux ASLR randomizes all objects per-exec, so once the process has been created, all subsequent objects (mmap requests) are located side by side. On the other hand, OS X only randomizes libraries per-boot and libraries addresses are shared among all executables even after they are relaunched.

Per-deployment: The application is randomized when it is installed in the system. This form of randomization, also known pre-linking, was proposed by Bojinov et al. [20] as a mechanism to provide randomization on systems that do not support position independent code (PIC) or relocatable code.

Per-boot: The randomization is done every time the system boots. That is, the random value or values used to map objects are generated a boot time (see Figure 1). This form of randomization is typically used on systems whose shared libraries are not compiled as PIC, and so the loader has to relocate the memory image to match the actual addresses. This technique for sharing libraries has some drawbacks:

- Once a library has been relocated in memory, it is no longer a copy of the file, but it has been modified to match the current virtual addresses where it has been loaded. Therefore, all subsequent requests to map that library shall use the same addresses in order to share the same pages.
- Since the memory image does not match the file image (because it has been customized for the current position) it is not possible to use the file itself as swap-in area of the image. A full swap-out and swap-in sequence on a swap device is necessary.

PIC code is implemented using relative addressing (the compiler emits offsets with respect to the program counter (PC) rather than absolute directions). Unfortunately, PC relative addressing modes are not available on the i386 architecture, which makes the code slightly larger and slower.

The effectiveness of ASLR has pushed some processor developers to include relative addressing in their new architecture families. The x86_64 architecture implements relative for 64 bits instruction set.

Per-exec: The randomization takes place when new executable image is loaded in memory (see Figure 1). In the literature, this form of randomization is known "pre-process randomization". But it must be pointed out that the randomization takes place when a new image is loaded (via the `exec()` system call) rather than when a new process is created by calling `fork()`.

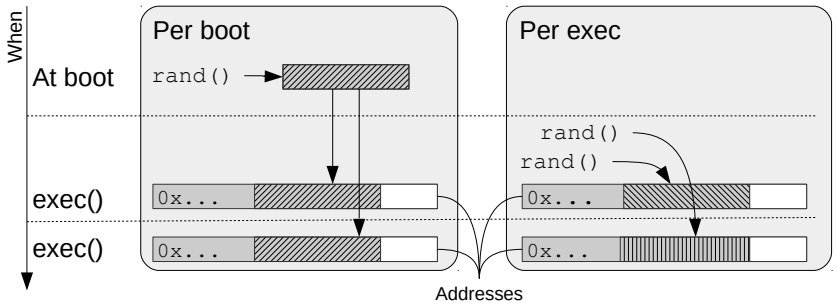


Figure 1. Per-boot versus Per-exec randomization.

Per-fork: The randomization takes place every time a new process is created (forked). Recently, Kangjie et al. [21] proposed *RuntimeASLR*, which implements an aggressive re-randomization of all the fine-grain objects of every child after `fork()`. This solution sets ASLR of the forked/cloned processes at the same level than the one achieved by an `exec()`. Unfortunately, Unix API defines that child processes inherit the memory layout of their parents, and so *RuntimeASLR* breaks compatibility.

It is still possible to have a per-fork randomization, while preserving API compatibility if randomization is only applied to new objects. Current ASLR designs allocate new objects in consecutive addresses, but it is possible to re-randomize the base address after a fork, so that new objects are unknown to parent and siblings.

Per-object: The object is randomized when it is created (see Figure 2). Note that, objects that are at a constant distance from another already mapped object are not considered to be randomized on a per-object basis, even if the reference object is randomized. Note that if the position of one of the two objects is leaked, then the position of the other is immediately known.

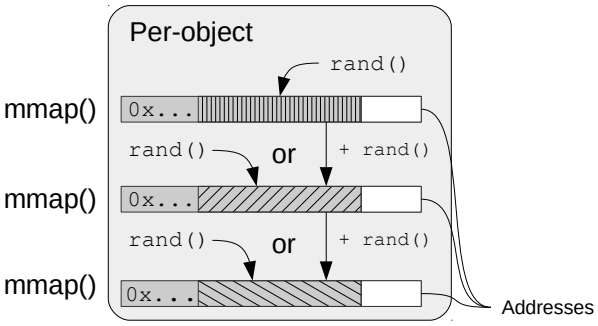


Figure 2. Per-object randomization.

2.2. Dimension 2: What

The second dimension determines the granularity of What is randomized. Whereby the more objects are randomized, the better. Some security researchers consider that if a single object (for example, the executable) is not randomized, ASLR can be considered broken. A more aggressive form of ASLR is when the randomization is applied to the logical elements contained in the memory objects: processor instructions [22], blocks of code [11,23], functions [12,24,25] and even to the data structures of the application [26]. In these cases, the compiler, the linker and the loader are required to play a more active role. Although these solutions are more advanced and secure, they are still not included in current systems.

2.3. Dimension 3: How

The way an object is randomized is defined by the last dimension: How. It is possible to consider two sub-dimensions: (1) how many bits are random and (2) what is the randomness between objects (inter-object). That is, the absolute entropy of the object by itself and the conditional entropy between objects. Regarding how many bits of the address are randomized, there are two forms: **full-VM** and **partial-VM**. Partial-VM is when memory space is divided in disjointed ranges to generate random numbers for the addresses. In Full-VM complete virtual memory space is used to randomize an object. The requirements required to carry out a full-VM are analyzed in the next sections. As far as we know, current ASLR implementations only use partial-VM randomization.

Partial VM: The virtual memory space is divided in ranges or zones (see Figure 3). Each zone contains one or more objects. Typically, zones do not overlap, and so, each zone defines a proper subset of the memory space. In most implementations, only a small range of the memory space is used, that is, the union of all the ranges does not cover all the virtual memory. Partial VM randomization greatly simplifies the implementation of ASLR because object collision are prevented, but the effective entropy is reduced.

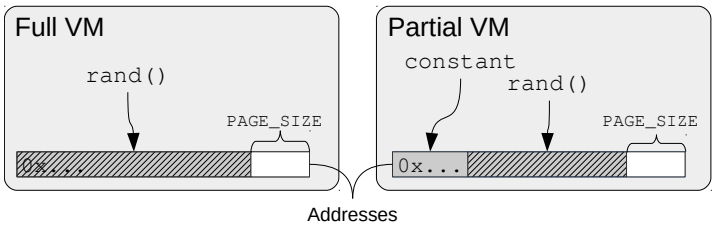


Figure 3. Full versus Partial virtual memory randomization.

Full VM: All the virtual memory space is used to randomize the objects (see Figure 3). When this randomization is used, the order of the main areas (exec, heap, libs, stack, ...) are no longer honored. Special care must be taken to avoid overlapping and collisions. The effective entropy is greatly incremented. As far a we know, no current ASLR design uses the full VM randomization.

Isolated-Object: The object is randomly mapped with respect to any other (see Figure 4). Some attacks rely on knowing the address of an object to exploit another one because there is a correlation between them [27,28]. In order to be effective, the correlation entropy of the isolated objects with respect to the rest of objects must be greater than the absolute entropy of each one. Therefore, an information leak of the position of an isolated-object gives no hint of the memory layout of the process but the leaked object.

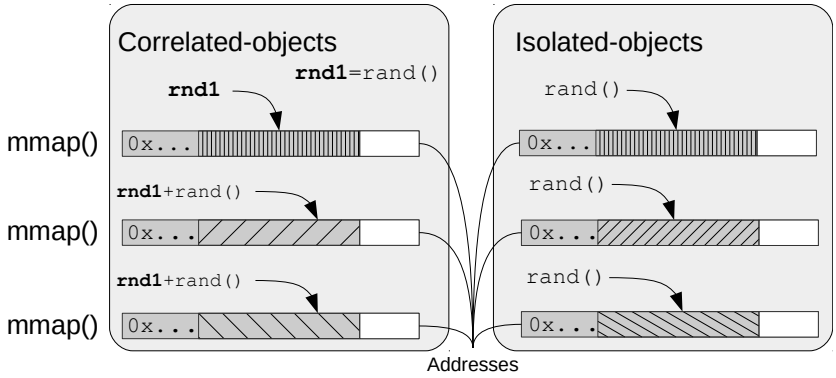


Figure 4. Correlated versus Isolated-object randomization.

Correlated-Object: The object is placed taking into account the position of another one (see Figure 4). The position of a correlated object is calculated as a function of the position of another object and a random value. When two objects are mapped together, side by side, they are fully correlated. Some examples of correlated objects are: Linux libraries and PaX thread stacks and libraries.

Sub-page: The offset bits of the page are also randomized (see Figure 5). By default, ASLR implementations use the processor virtual memory paging support to randomize objects. If no additional entropy is added, addresses are page aligned. Depending on the type of object (shared object, contains data or code, swap constraints, etc.) sub-page randomization may be implemented transparently. For example, the stack and the heap have sub-page randomization in current Linux.

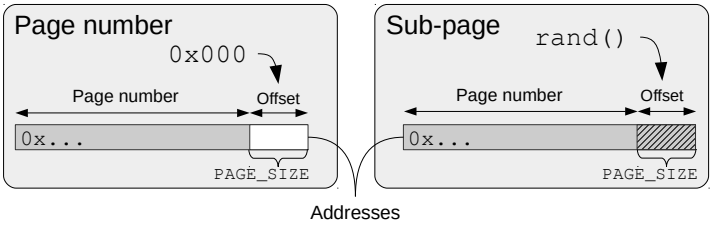


Figure 5. Sub-page randomization.

Direction: Up/down search side used on a first-fit allocation strategy (see Figure 6). When allocating objects together, new objects can be placed at higher addresses that the ones already mapped (bottom-up) or at lower addresses (top-down). The direction is used to decide the side to place new objects. There are two situations when the direction is necessary:

- When objects are randomly mapped, it may occur that (if not prevented) a randomly generated address collides with an existing object, in this case, the direction determines the side of the existing object (up or down) where the new one will be mapped.
- When objects are mapped together: the direction bit determines how the area or zone grows.

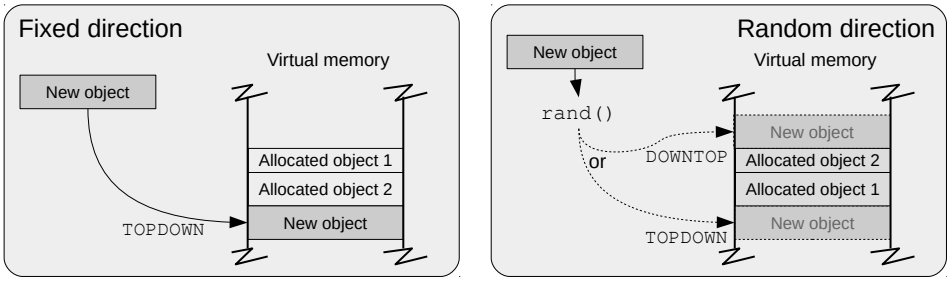


Figure 6. Direction of allocation.

Note that the direction is not a global parameter or feature of the ASLR, but its scope can be determined on a per-object or per-zone basis.

Bit-Slicing: The address of an object is the concatenation of two, or more, random numbers which are obtained at different times (for example, at boot and at exec), as shown in Figure 7. For example, this form can be used when a subset of the addresses must be aligned to a fixed value because of performance reasons. In this case, the alignment can be randomly chosen at boot time, and then align all the mappings to that value. Later, all address bits may be randomized except the aligned bits which are set to the value chosen previously.

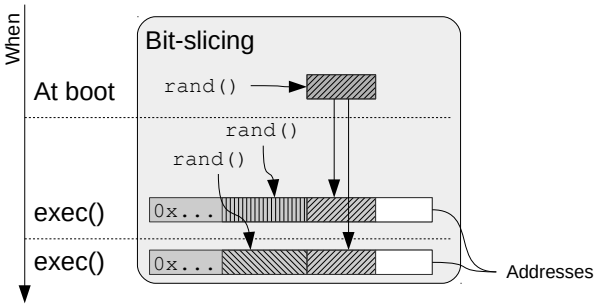


Figure 7. Bit-Slicing randomization.

Specific-zone: A base address and a direction where objects are allocated together. Taking into account security aspects, the more random bits and the more independent are the mappings the better. On the other side, if the goal is to reduce the overhead, the more compact the layout the better. Specific-zones defines a mechanism that can be used to group together objects of the same or similar level of security, and isolate each group from others of different level of security/criticality Figure 8 shows an example of how the objects allocated to an specific-zone are mapped.

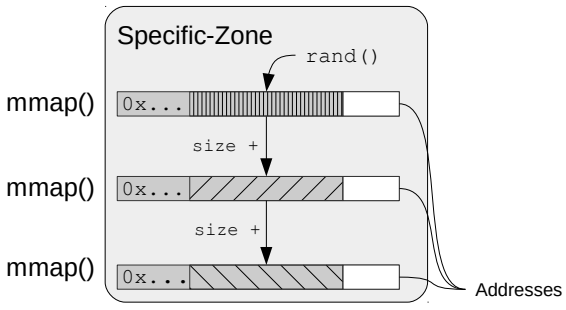


Figure 8. Specific-zone example.

Table 1 summarizes all form of randomization grouped by the dimensions **When**, **What** and **When**. Those are the fundamental features to assess the quality and robustness of ASLR implementations.

Table 1. Summary of randomization forms and dimensions.

Feature		Description
When	Per-boot	Every time the system is booted.
	Per-exec	Every time a new image is executed.
	Per-fork	Every time a new process is spawned.
	Per-object	Every time a new object is created.
What	Stack	Stack of the main process.
	LD	Dynamic linker/loader.
	Executable	Loadable segments (text, data, bss, ...).
	Heap	Old-fashioned dynamic memory of the process: `brk()`.
	vDSO/VVAR	Objects exported by the kernel to the user space.
	ARGV	Command line arguments and environment variables of the process.
	Mmaps/libs	Objects allocated calling `mmap()`.
How	Partial VM	A sub-range of the VM space is used to map the object.
	Full VM	The full VM space is used to map the object.
	Isolated-object	The object is randomized independently from any other.
	Correlated-object	The object is randomized with respect to another.
	Sub-page	Page offset bits are randomized.
	Bit-slicing	Different slices of the address are randomized at different times.
	Direction	Top-down/bottom-up search side used on a first-fit allocation strategy.
	Specific-zone	A base address and a direction where objects are allocated together.

The presented randomization forms are analyzed in Sections 5 and 8, to properly assess the ASLR effectiveness of Linux, PaX and OS X. This taxonomy is key not only to assess current ASLR effectiveness but also to develop future ASLR designs and implementations, specially in 32-bit systems were the available virtual memory map introduces a challenge to all ASLRs.

3. ASLR Limitations

In this section we analyze the problems and limitations present in all ASLR implementations employed by modern operating systems. The main limitation is introduced by the fact that there are objects (e.g., `stack`, `heap`) that need to grow in memory at runtime. This requires to divide the virtual memory region to ensure that those objects can grow.

In more detail, the Linux and PaX (FreeBSD, HardenedGentoo and others use the PaX ASLR approximation) ASLR designs rely on the same core ideas, in that they define four partial-VM areas: (1) stack, (2) libraries/mmaps, (3) executable and (4) heap. The classic memory layout was designed by considering that some zones or objects are *growable* (main stack, thread stacks and heap). Ideally, a growable object is a contiguous area of memory which can be expanded or shrunk dynamically according to the needs of the program. In order to allow them to grow, they are typically placed in the border (top or bottom) of virtual memory, far away from other objects, otherwise they will not grow or, even worse, a silent collision could occur.

Originally, the functionality of growable objects was a smart, simple and efficiently solution for efficient memory usage. However, the use of threaded applications and the possibility of adding dynamically new objects into the memory space forced developers to reconsider the viability of these growable areas. Today, growable objects are a source of numerous problems [29,30], but fortunately a set of advanced programming solutions has been developed.

Growable objects impose strong limitations on ASLR design, and they affect negatively the entropy of all objects. The situation gets even worse when multiple growable objects are used in the application, as actually happens with multiple thread stacks. The approach used in Linux (see Figure 9) involves placing each object as separately as possible from each other, which forces to fix high bits of the addresses, thus degrading effective entropy. Since the extremes of the virtual memory

are already occupied, libraries and mmaps are placed between the stack and the heap. Note that a small shared library is automatically mapped by the kernel into the address space of all user-space applications (vDSO). Therefore, both static and dynamically linked (PIE or not) programs have a similar memory layout.

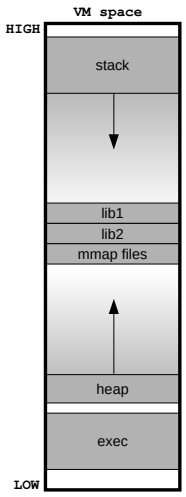


Figure 9. Classic memory layout.

Originally, PIE-compiled applications were loaded jointly with the libraries, but after the Offset2lib weakness [27] was identified, the PIE executable was moved to lower addresses (Linux 4.1) in its own zone.

3.1. Stacks

There are two different kinds of stacks, namely the stack of the main process, and the stack of the threads or clones (since both thread and cloned entities handle stacks in a similar way, in what follows we will refer to them jointly as 'thread stacks'). The main stack is still considered and handled as a growable object.

Initially, thread stacks were 'set up' to be growable. Flags `MAP_GROWSDOWN` and `MAP_GROWSUP` were added to the `mmap()` request, to tell the kernel about expected behavior. Inevitably, these flags were removed [29] because of security problems and intrinsic logical limitations. Nowadays, thread stacks are treated as regular (non-growable) objects, reserved with the maximum estimated size when the threads are created. By default, the thread stack size is set to 8 Mb (in Debian and Ubuntu), but it can be changed by using the `RLIMIT_STACK` resource with the `setrlimit()` system call. Note that the `RLIMIT_STACK` value is used as the default thread stack size rather than an upper limit.

A summary of the facilities provided by the system (compiler and standard library), to deal with growable stack issues, is presented below:

- One or more protected pages (page guards) are placed at the end of the thread stack. If the stack overflows, then the process receives a SIGSEGV signal. This guard is further enforced by the GCC flag `-fstack-check`, which emits extra code to access sequentially all the pages of the stack, thus preventing overflowing by jumping over the page guard.
- The split stack feature (GCC flag `-fsplit-stack`) generates code to automatically continue the stack in another object (created via `mmap()`), before it overflows. As a result, the process has discontinuous stacks which will only run out of memory if the program is unable to allocate more memory. This is an interesting feature for threaded programs, as it is no longer necessary to calculate the maximum stack size for each thread. This is currently only implemented in the i386 and x86_64 back-ends running in GNU/Linux.

- It is possible to ask the compiler to print stack usage information for the program, on a per-function basis, using the `-fstack-usage` flag and making an estimation of stack size.

Although these facilities are very helpful when dealing with stacks, in practice most applications work with the default 8Mb sequential stack (Google Chrome(r), LibreOffice, Firefox, etc.). Only very demanding applications have stack size issues, which are typically handled by slightly increasing the `RLIMIT_STACK` limit value. For example, Oracle(r) advises to set it to 10MB when running its database.

3.2. Heap

When the process needs more heap memory, it calls the `brk()` system call to move forward (higher addresses) the heap's end. The operating system tries to expand sequentially the heap object to provide the requested memory. The `brk()` request may fail if (1) there is not enough free memory contiguous to the existing heap, because the end of the memory has been reached or because another object is already in that address, or (2) the data segment limit has been exceeded, as set by the `RLIMIT_DATA` resource.

The heap is used by the standard C library to provide dynamic storage allocation (DSA): `malloc()` and `free()`. Although originally DSA algorithms relied exclusively on heap memory, current implementations use multiple non-contiguous objects of memory requested by `mmap()`. In fact, the GNU libc uses `mmap()` when the size is larger than 128 Kb. Also, `brk()` was marked as LEGACY in SUSv2 and removed in POSIX.1-2001.

4. ASLRA: ASLR Analyzer

In this section we present the tool we have developed to effectively assess ASLR implementations. ASLRA discovered several weaknesses, as Section 5 describes, and also assessed the effectiveness of the proposed ASLR-NG in Section 7.

Although it is possible to analyze how ASLR has been implemented in the operating system to calculate the intended or expected entropy, there are too many interactions and adjustments between the code that generates the random values and code that finally assigns the address to the object. In fact, we have detected several security issues by observing the external entropy of both Linux and PaX [31–33].

Peter Busser developed a tool called `paxtest` (included in most Linux distributions) to estimate, among other security features, the entropy of memory objects. It uses a custom *ad hoc* algorithm to guess the effective entropy bits. This algorithm has been designed assuming that the underlying distribution is uniform with a power of 2 range. When these conditions do not hold, the result is incorrect. Also, it does not provide basic statistical information about the observed distribution. For example, PaX suffers from non-uniform weaknesses (see Section 5.2) on most mappings, and so it is overestimated by the paxtest tool.

We have developed ASLRA, a test suite, which can be used to measure and analyze the entropy of all the objects. ASLRA is composed of two separated utilities:

Sampler: Launches a large number of processes and collects the addresses of selected memory objects.
Analyzer: The resulting sampled file is processed to calculate several statistical parameters (see Figures 10–12). Besides the basic ones: range, mean, median mode and standard deviation, the analyzer calculates four different entropy estimators: (1) flipping bits, (2) individual byte Shannon entropy, (3) Shannon entropy with variable width bins and (4) Shannon 1-spacing estimator [34]. The tool also provides information about memory fragmentation, conditional entropy, and multiple plots (histogram, distribution, etc.).

We have used the term "entropy" in this paper to refer to the amount of randomness exhibited by an object. In information theory, entropy is used as a measure of the amount of information provided by a piece of data, which typically is the probability occurrence. For our purposes, entropy

shall be defined as "*the amount of uncertainty that an attacker have about the location of a given object*". Shannon entropy is formally defined as:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x)$$

The resulting value is a good measure of the dispersion or surprise, but it must be interpreted with caution [35]. In most cases, it is an accurate estimation of the cost of an attack, but only if it is a uniform distribution. It is especially problematic for those distributions with a high kurtosis, because the attacker can focus on a small range of values, thereby building faster attacks.

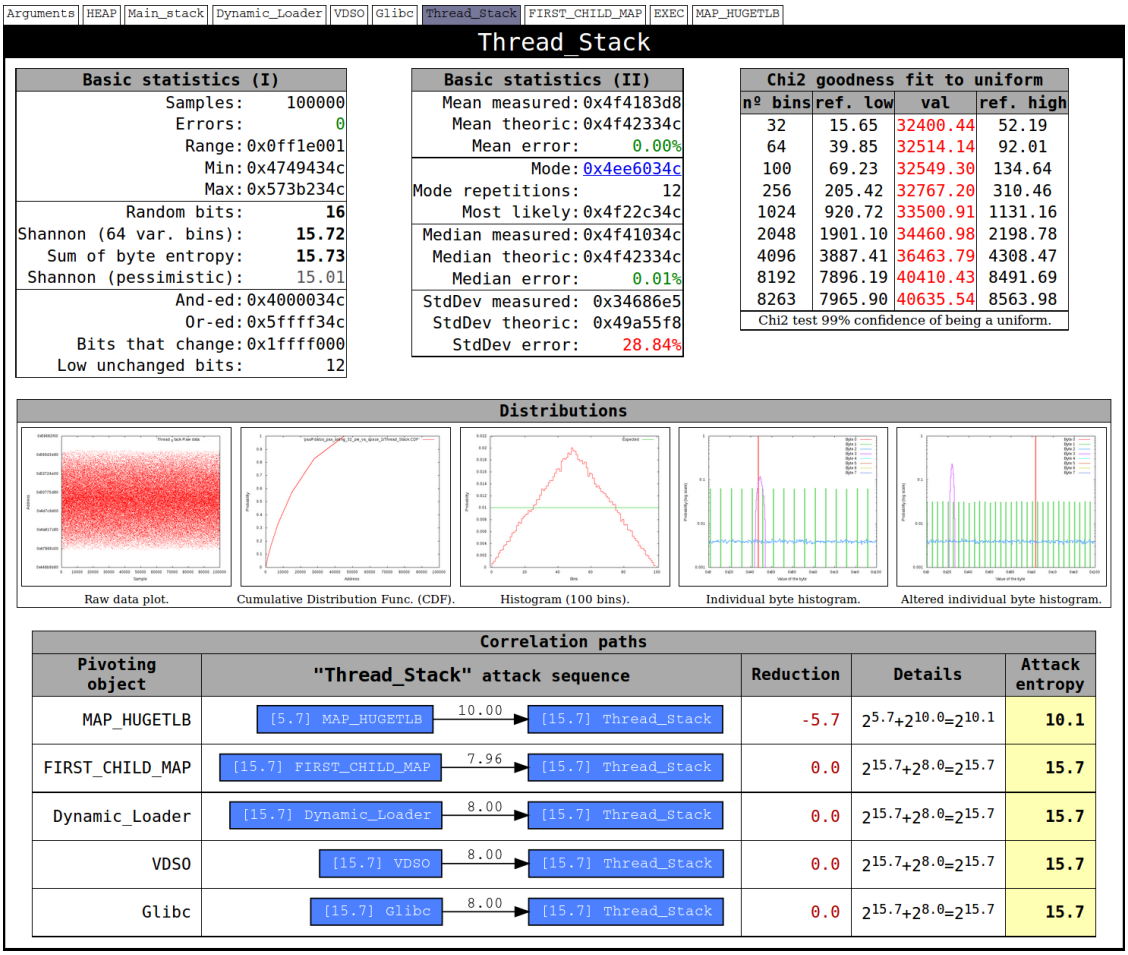


Figure 10. Statistical information produced by ASLRA for the Thread stack on PaX i386.

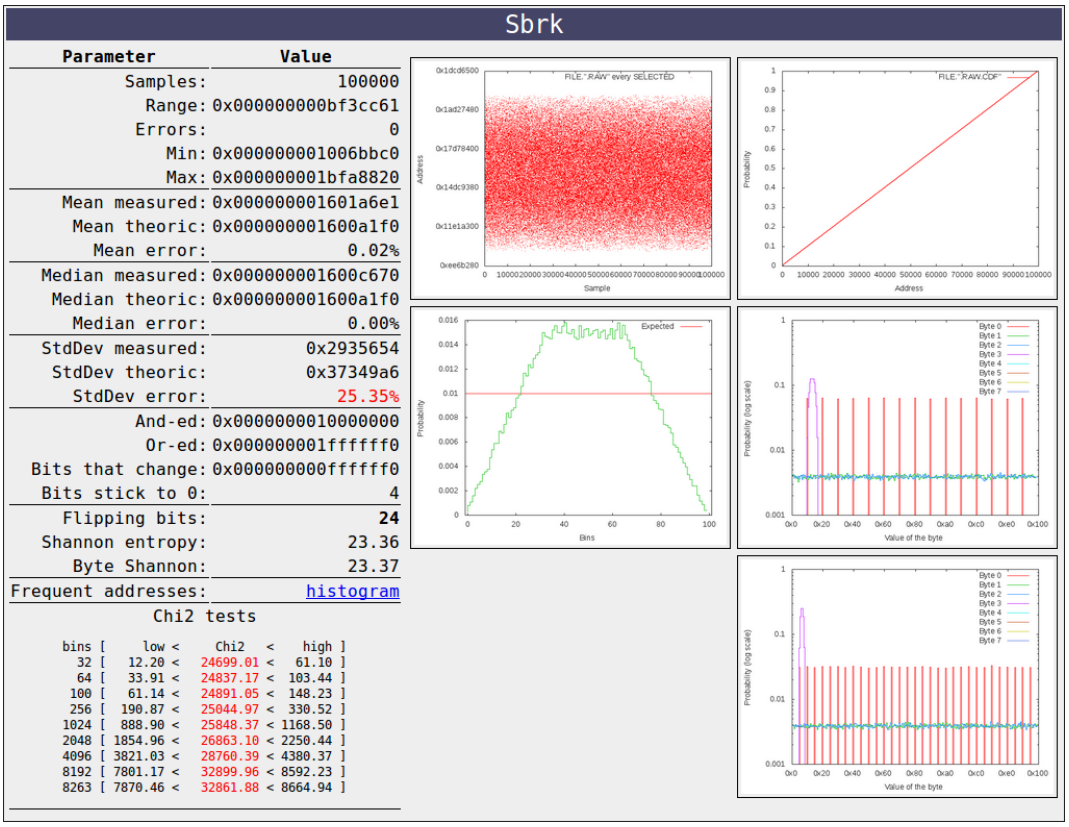


Figure 11. ASLR analyzer: Screenshot of PaX Heap (brk) on i386.

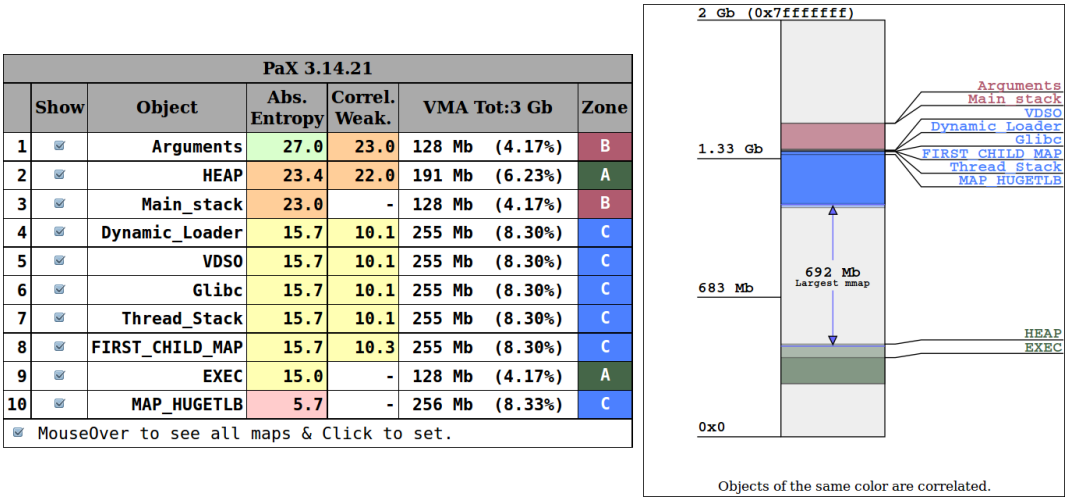


Figure 12. ASLRA: summary of PaX ASLR in a 32 bit system.

Entropy estimation is a challenging issue [36]. In order to obtain accurate results, it is necessary to have very large data sets or alternatively make assumptions about the underlying distribution. The sampler part of the ASLRA tool is a simple application using `malloc()`, `mmap()`, etc., and has been optimized to generate millions of samples in a few minutes to avoid biased estimations for most distributions. If single variable entropy is a challenging task, measure the correlation between objects and properly estimate the conditional entropy requires a huge set of samples. We have addressed the issue by assuming that the relation between objects are defined by a sum of a uniform random variable or a constant value. Therefore, the conditional entropy is calculated as the entropy of the difference of the objects. Table 2 shows the memory objects that the developed ASLRA tool can collect.

Table 2. Objects analyzed by the ASLRA tool.

Object	Description
Arguments	The arguments received by `main()` and the environment variables of the process.
HEAP	The initial heap location as returned by `brk()`.
Main Stack	The stack of the process, that is the address of a local variable of the `main()` function.
Dynamic Loader	For dynamic executables, the address of `ld.so`.
vDSO	Linux specific object exporting services like for example the syscall mechanism.
Glibc	The standard C library used by the majority of processes.
Thread Stack	The stack created by default then a new thread is created by the `libpthread.so` library.
FIRST_CHILD_MAP	The address returned by the first `mmap()` object of a child process.
EXEC	The address where the executable is loaded.
MAP_HUGETLB	Address of a large block (2 Mb) reserved via `mmap()` using the flag `MAP_HUGETLB`.

The resulting conditional entropy of the system is presented using a table and a graph that shows the absolute entropy and the coentropy. See Figure 13.

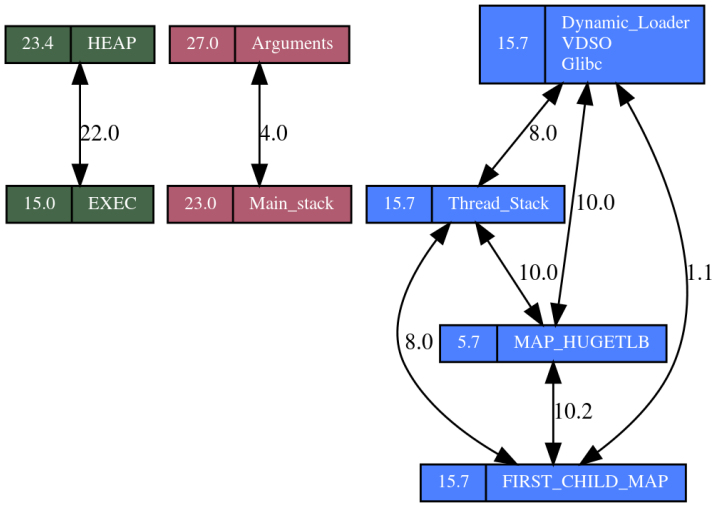


Figure 13. ASLRA: Correlation between objects of PaX in a 32 bit system.

5. ASLR Weaknesses

In this section we describe four ASLR design weaknesses that affect current ASLR implementations. The result of these weakness are represented in Table 4 as a lack of randomization forms, and as a low absolute entropy in Table 5.

5.1. Non-Full Address Randomized Weakness

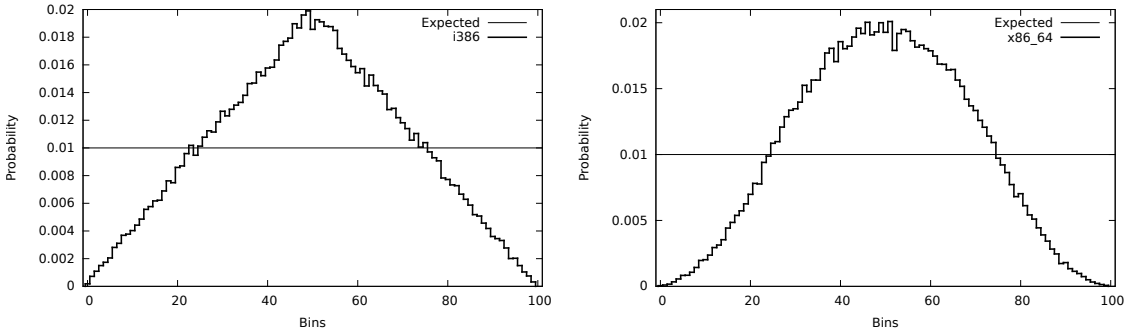
ASLR has been implemented by slightly shaking or moving randomly the base address of objects with respect to the classic layout, where the main stack is at the top, the executable is at the bottom, the heap follows the bss segment and the mmap zone is located in between the heap and the stack. Therefore, the entropy that can be applied to each object is limited by the range that they can be moved while preserving the previous sequence. This affects the higher bits of the addresses [6].

Another constraint that reduces entropy is the unnecessary alignment of some objects. Although alignment is mandatory in some objects (huge pages, executable, libraries, etc.), others like stacks, heap have sub-page. It may be possible to extend the sub-page to more objects by extending the semantic of the `mmap()` syscall.

5.2. Non-Uniform Distribution Weakness

The distribution of objects along the range should be as uniform as possible. That is, all the addresses should have the same, or very similar, probability of occurrence; otherwise, it would be possible to speed up attacks by focusing on the most frequent (likely) addresses.

Figure 14 shows the output of the ASLRA (ASLR analyzer) tool for the libraries and mmap objects in PaX. The distributions of these objects in i386 follows a triangular distribution and on x86_64 an Irwin-Hall with $n = 3$. In Linux, the heap is the result of the sum of two random values, but since one of them is much larger than the other, the impact on the distribution is almost negligible.



(**a**) On a i386 system (**b**) On a x84_64 system

Figure 14. Statistical distribution of mmapped (libraries, thread stacks, anonymous maps, etc.) objects using PaX security mode.

5.3. Correlation Weakness

Attacks launched to bypass ASLR are becoming more and more sophisticated; for instance, instead of attacking an object directly, the attacker can first de-randomize an object with low entropy, and then use it to de-randomize the target object (the object which contains the required gadgets or data). The idea that an object's memory address leak can be used to exploit another one was first demonstrated by Marco-Gisbert et al. [27] through the offset2lib weakness. In that case, the executable was de-randomized using a byte-for-byte attack [37] and then the libraries zone was calculated, resulting in a very fast bypass of ASLR. This weakness was analyzed by Herlands, W at al. [38], and referred as EffH (Effective Entropy).

In Linux and PaX, the heap and the executable are separated from each other by a random value. A leak in the heap area does not only compromises the heap, but it also reveals information about the executable, because there is less entropy distance from the executable to the heap (correlation entropy) than the absolute entropy of the executable. Huge pages and the objects in the mmap_area are also correlated. Since huge pages have the largest alignment, they have the lowest entropy, and attackers can build correlation attacks by de-randomizing huge pages first and then later the libraries zones. For example, in PaX i386, instead of attacking the libraries directly (16 bits), attackers can de-randomize huge pages (6 bits) and later de-randomize the libraries zone from the huge page zones (10 bits). This two-step attack takes $1088 = 2^{10} + 2^6 \approx 2^{10}$ attempts, instead of the 65,536 = 2^{16} tries required to directly attack the library.

All mmapped objects are located together in the mmap_area, which results in total correlation between them. From a security point of view, this is a weak design. MILS (Multiple Independent Levels of Security/Safety) [39] criteria state that objects of different criticality levels must be isolated. Google Chrome, for instance, is aware of this security issue and has addressed it by implementing some form of user-land ASLR to map JIT (just-in-time compiled code) objects in its own zone, which is separated from the default mmap_area. Note that JIT objects are an appealing target for attackers [40].

5.4. Memory Layout Inheritance Weakness

All child processes inherit/share the memory layout of the parent. This is the expected behavior, since children are an exact copy of the parent's memory layout. Unfortunately, though, from a security point of view, this is not a desirable behavior because although new objects belong only to the child process, their addresses can be guessed easily by parents and siblings. This problem is especially dangerous on networking servers which use the forking model. In Android, for instance, the situation is even worse, because all of the applications are children of Zygote, and although the siblings might not call the same mapping sequence, a malicious sibling can predict future mmaps of any other. Therefore, the leakage of an object in the library or mmap area exposes all objects in these areas (correlation weakness) and also allows one to predict where future mmaps will be placed-even between siblings (inheritance weakness).

This issue is widely known. The solution used by the SSHD suite consists on relaunching (fork + exec) the process for every incoming connection, rather than creating a direct fork process. In this way, not only the new maps are different among siblings, but all the maps are different. Lee et al. [41] proposed to use the same solution to replace the application creation model of Zygote by a pool of pre-exec processes, this new model was called Morula.

6. ASLR Constraints and Considerations

The straightforward solution to solving most of the previous weaknesses is to randomize each object independently over the full VM range. Although this idea is quite intuitive [12], multiple practical issues must be addressed properly, in order to achieve a realistic ASLR design. ASLR-NG has been designed by taking into account the following issues:

Fragmentation: although, from the point of view of security, having objects spread all over the full VM space is the best choice, in some cases it introduces prohibitive fragmentation, which is especially severe in 32-bit systems. Applications that request large objects or make a lot of requests may fail randomly, so it is mandatory to have a mechanism to address this fragmentation.

Page table footprint: a very important aspect that is underestimated is the size of the process page table, because the more the objects are spread, the bigger the page table. This is particularly important in systems with low memory or with a high number of processes. Since each application could have a different level of security, the ASLR design should allow for tuning the page table size versus object spreading.

Growable areas: unfortunately, most applications still use growable areas in some objects. In order to be compatible with these applications, an ASLR must guarantee some form of compatible behavior.

Homogeneous entropy: all objects should have the same amount of entropy, in particular objects of the same type (for example, stacks); otherwise, attackers will focus on the weakest link. Unfortunately, none of the current designs meets this requirement.

Uniformly distributed: all objects should be uniformly distributed; otherwise, attackers can design more effective attacks by focusing on the most frequent addresses.

ASLR compatibility: the ASLR design should be backward-compatible with existing applications. That is, if there is a trade off between security and compatibility, then the design should allow for tuning the application framework to meet application's needs.

7. ASLR-NG: Address Space Layout Randomization Next Generation

This section describes the proposed ASLR-NG, which addresses all the weaknesses identified in Section 5 as well as all of the constraints and considerations presented in Section 6.

ASLR-NG does not divide the virtual memory region enabling to load any object (stack, heap, libraries, etc.) at any address without any restriction. In order to achieve this, ASLR-NG limits and pre-reserve (no actual memory is being allocated), all growable objects (main stack and heap).

When those objects need to grow, they will consume more part of the pre-reserved memory until they reach the limit. The pre-reserved memory area can be seen as a memory belonging to a particular object where others objects can not be allocated.

As presented in Section 3, both the stack and the heap are growable but limited, which makes ASLR-NG a realistic and practical ASLR. ASLR-NG uses those limits to create a pre-reserved memory for each growable object. This ensures compatibility with applications since by adjusting those limits. In fact by pre-reserving memory, ASLR-NG prevents accidental mappings and collisions. For example in current implementations it is possible to `mmap()` an object very close to the stack resulting in a collision when the stack grows but this is not possible with ASLR-NG since only the stack can allocate memory to its pre-reserved area and the `mmap()` will fail.

7.1. Allocating Object Strategy

Two methods are available to allocate an object in ASLR-NG: Isolated and in an specific-zone.

- **Isolated:** the object is independently randomized using the full virtual memory space of the process. Unlike current implementations, ASLR-NG can use the full VM range to allocate an object, and as a result there no order to the objects and it prevents any kind of correlation attack.
- **Specific-zone:** objects of the same class are mapped together and isolated from others. A specific-zone is defined by a base address and a direction flag, both of which are initialized when the specific-zone is created (see function `new_zone()` in Listing 1). The base address is a random value taken from the full VM space, and new objects are placed by following the direction flag (toward higher or lower addresses) with respect to the base address.

  The main benefit of using specific-zones is that it reduces both fragmentation and page table footprint, which makes ASLR practical and realistic. Furthermore, specific-zones can be created according to MILS criteria, in that objects of the same criticality level may be grouped together. Criticality depends, among other factors, on the permissions and the kind of data stored on the object. Following this rule, ASLR-NG defines five specific-zones (depending on the configuration, see profile modes below):

  Huge pages: placing all huge page objects in their own specific-zone removes the correlation weakness between huge pages and normal mmapped objects. This is a specially dangerous form of correlation weakness as described in Section 8.3.
  Thread stacks: following the same criteria as the main stack, the thread stacks are isolated from the rest of objects on their own specific-zone.
  Read-write-exec objects: although these types of object are seldom used, for example in JIT mapping, they are very sensitive; in fact, Google implements custom randomization in their Chromium browser for these objects as part of its sand-boxing framework.
  Executable objects: map requests with executable permission are grouped in a specific-zone. This zone is mainly used to group library codes.
  Default zone: any other objects that do not match previous categories are allocated to this specific-zone. In addition, applications can create custom specific-zones to isolate sensitive data. For example, the credentials or certificates of a web server can be isolated from the rest of the regular data. This mechanism can prevent a Heartbleed [42] attack by moving sensitive data (certificates) away from the vulnerable buffer.

7.2. Addressing Fragmentation

When virtual memory size is small, fragmentation is an issue, because the more objects that are independently randomized, the more fragmented the memory. In dynamic memory, the fragmentation problem is defined as [43] "the inability to reuse memory that is free."

There is no simple way to measure fragmentation, but the worst case depend on: (1) the number of objects already allocated, (2) their size, (3) the relative position of each one and (4) the size of the

new request. If all objects, n, are independently randomized, the worst case occurs when the allocated objects are of one page size and they are evenly distributed along the whole memory space. In this case, the maximum guaranteed size is approximated by:

$$\text{new\_obj\_size} \lesssim \frac{\text{VM\_SIZE}}{n+1}$$

On x86_64 fragmentation is not a issue because of the very large number of mapped objects needed to cause an error. For example, a 1GB memory request will not fail until 2^{17} = 131,072 objects have been mapped.

On the other hand, fragmentation is a real problem in 32-bit systems. For example, a memory request of 25MB is not guaranteed after just 122 requests (of page size), while a request for 256 MB may fail after mapping just 12 objects, including the stack, vDSO, executable, heap, each library, etc. Therefore, it is not practical to randomize each object independently in 32-bit systems, without addressing the fragmentation issue.

ASLR-NG addresses this issue by reserving a range of virtual space, the amount of which is specified as a percentage of the available VM size. When a requested object does not fit into the non-reserved space, the allocation algorithm automatically uses the reserved space, without degrading the entropy of these objects and regardless of their size.

Figure 15 shows the result of allocating multiple objects in ASLR-NG. Objects 1 and 3 fit into the non-reserved area, and so they are placed there, but for objects 2 and 4, there are no free gaps to hold them on the non-reserved area. In this case, the algorithm performs a top-down, first-fit strategy. Note that objects allocated in the reserved area will 'inherit' the entropy of the lowest object in the non-reserved area.

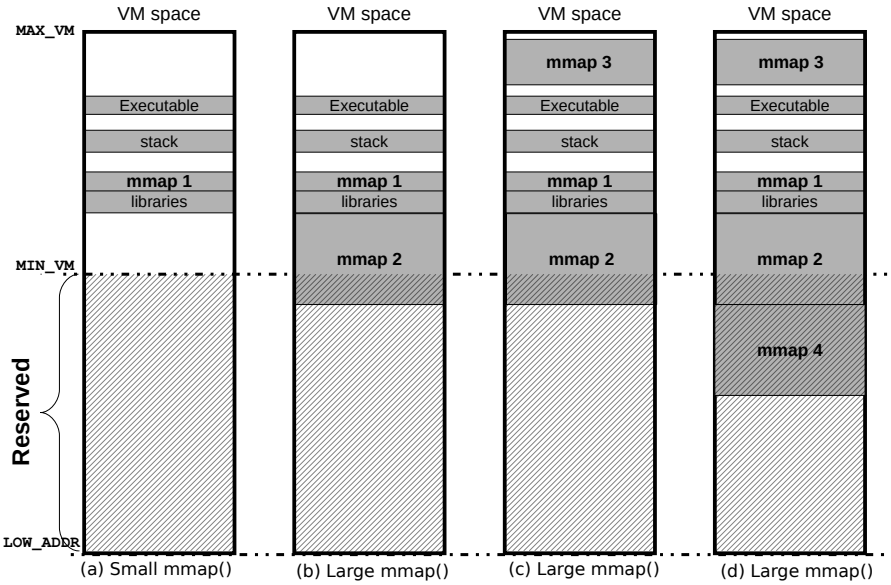


Figure 15. ASLR-NG: A 50% example of a reserved area.

Although reserving a percentage of the VM will reduce the range for available randomization, ASLR-NG uses a novel strategy to regain lost entropy, whereby the reserved area is randomly placed at the top or the bottom of the virtual memory space. For example, by reserving 50%, an attacker cannot know on which side (top or bottom) the objects will be located, which forces them to consider the whole VM space. As a result, there is no entropy penalty with this strategy. Only when the reserved area is larger than 50%, is there a small amount of entropy degradation. The expression which relates the loss of entropy to the percentage of reserved area is:

$$f(x) = \begin{cases} 0, & \text{if } x \leq 50\% \\ -1 - \log_2\left(1 - \frac{x}{100}\right), & \text{otherwise} \end{cases}$$

where x is the percentage of the reserved area, and $f(x)$ gives the number of bits that have to be subtracted from the total VM space entropy. For example, reserving 50% on an i386, the largest guaranteed object is 1.5 GB and entropy is not reduced. If 2/3 of the VM space is reserved, then it is possible to allocate an object up to 2GB in size, and at the cost of reducing entropy by only 0.5 bits. Therefore, the ASLR-NG design has both more entropy and less fragmentation.

7.3. Algorithm

When a process is created, the area reserved to avoid fragmentation is defined by setting the variables min_ASLR and max_ASLR. This is the range that will be used to allocate objects (allocation area).

The direction of a specific-zone is a random bit with a probability of pointing towards the middle of the allocation range inversely proportional to the distance of the base address to the middle—the expression is in Listing 1. In other words, if the base address is close to the border of the allocating range, then the direction is more likely to point toward the other side of the range. This way, objects will not accumulate at the borders of the allocation area.

Listing 1. ASLR-NG initialization pseudo-code.

```
new_zone(low, high, zone) {
zone.base     = randomize_range(low, high);
zone.direction = randomize_range(low, high) < zone.base ? TOPDOWN : DOWNTOP;
}
do_exec(){
...
reserved =  VM_SIZE * percentage_reserved / 100;
min_ASLR = reserved * (rand() % 2);
max_ASLR = min_ASLR + VM_SIZE - reserved;
new_zone(min_ASLR, max_ASLR, mmap_base);
new_zone(min_ASLR, max_ASLR, huge_pages);
new_zone(min_ASLR, max_ASLR, thread_stacks);
...
}
```

A detailed analysis of the distribution of the objects at the borders of the allocation area is beyond the remit of this paper, but for now we can say that the presented algorithm to determine the direction gives a fair distribution along the whole range, with no accumulation areas (addresses with higher probability), regardless of the number of objects in the zone and the workload mix.

The algorithm employed to allocate an object works by first selecting a value as a hint address, in order to place the object, and then to look for a free gap in which to actually place the object. The algorithm is as follows:

1. Obtain the hint address and the direction:
   - if it is to a specific-zone, then the hint address and the direction are the ones from the specific-zone.
   - if it is an isolated object, then the hint address is a random value from the allocation range [min_ASLR, max_ASLR] and the direction is top-down.
2. Look for a gap large enough to hold the request from the hint address to the limit of the allocation area determined by the direction. If found, then succeed.

3. Look for a gap large enough to hold the request from the hint address to the limit of the allocation area determined by the reverse direction. If found, then succeed.
4. Look for a gap large enough to hold the request from the full VM space, starting from the allocation area and working towards the reserved area. If found, then succeed.
5. Out of memory error.

Even if there is no reserved area, step 4 is necessary to guarantee that the whole virtual memory is covered properly. For example, as illustrated in the Figure 16d, the gaps [ld.so ↔ mmap_base] and [mmap_base ↔ vDSO] are not suitable for a large request, but the gap [ld.so ↔ vDSO] can be used if a global search is done.

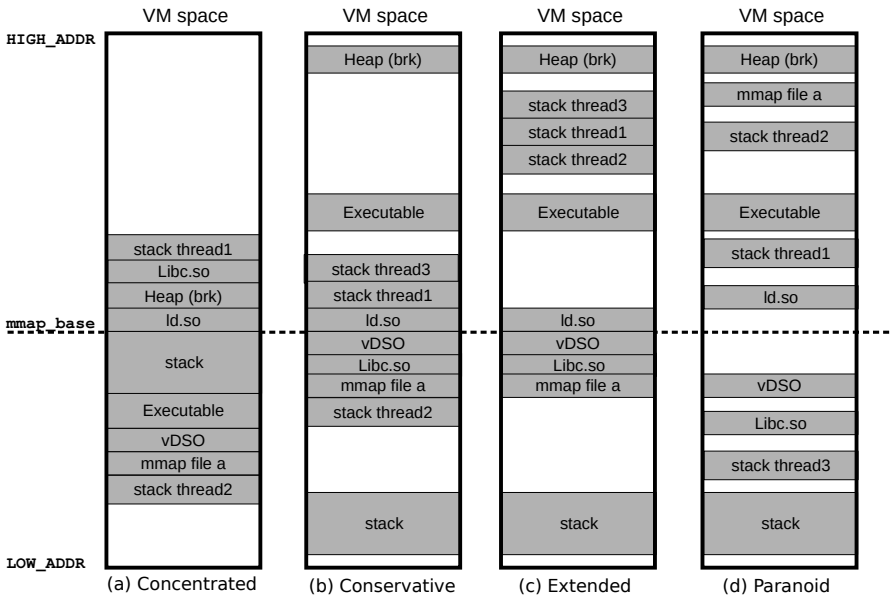


Figure 16. ASLR-NG: Profile mode examples.

7.4. Profile Modes

The ASLR-NG design provides two possibilities for allocating each object: isolated or in a specific-zone. From a security point of view, the more isolated objects, the better, but there are multiple side effects that should be carefully considered and balanced, as described in Section 6. In order to simplify the configuration of ASLR-NG, we provide four different working modes or profiles. Each mode randomizes each object using the isolated or the specific-zone method. The four modes are summarized in Table 3, and a representative example of each one is sketched in Figure 16. Next is the design rationale for each mode:

Mode 1—Concentrated: all objects are allocated in a single specific-zone, which results in a compact layout. The number of entropy bits is not degraded but only the correlation entropy between them. In other words, the cost (if brute force were used) to obtain the address of an object is not reduced by using this mode. The goal is to reduce the footprint of the page table.

Mode 2—Conservative: this mode is equivalent to that used in Linux and PaX. The main stack, the executable and the heap are independently randomized, while the rest (libraries and mmaps) are allocated in the mmap specific-zone. Since the objects are randomized using the full allocation range, ordering is not preserved; for example, the stack may be below the executable.

Mode 3—Extended: this is an extension of the conservative mode, with additional randomization forms: (1) specific-zones for sensitive objects (thread stacks, heap, huge pages, read-write-exec and only executable objects); (2) sub-page randomization of the heap and thread stacks and (3) per-fork randomization. This can be considered a very secure configuration mode

which addresses most of the weaknesses and sets a reasonable balance between security and performance. Therefore, this should be the default mode on most systems.

Mode 4—Paranoid: every object is independently randomized, and no specific-zones are used. As a result, there is no correlation between any objects, which could even prevent future sophisticated attacks. It is intended to be used on processes that are highly exposed, for example networking servers, but should be carefully used when applied globally to all system processes because of additional memory overheads.

Table 3. ASLR-NG mode definition.

Feature	Modes			
	1	2	3	4
Sub-page in ARGV	✓	✓	✓	✓
Randomize direction	✓	✓	✓	✓
Bit-slicing	✓	✓	✓	✓
Isolate stack, executable and heap		✓	✓	✓
Specific-zone for huge pages		✓	✓	
Randomize specific-zones per child			✓	✓
Sub-page in heap and thread stacks			✓	✓
Specific-zone for thread stacks			✓	
Specific-zone for read-write-exec objects			✓	
Specific-zone for exec objects			✓	
Isolate thread stacks				✓
Isolate LD and vDSO				✓
Isolate all objects				✓

7.5. Fine Grain Configuration

Each profile mode is defined by a set of features. Table 3 lists the ASLR-NG configuration options enabled on each mode.

- Sub-page in ARGV: ASLR-NG randomizes all the sub-page align bits. Although the arguments/environment are in the stack area, the page align bits of ARGV can be randomized.
- Randomize direction: the direction of a specific-zone is re-randomized for every new allocation. As a result, even libraries that typically are loaded sequentially will have some degree of randomness, which is especially useful in the concentrated profile, because it shuffles objects.
- Specific-zone for huge pages: if enabled, ASLR-NG uses a different specific-zone to map huge pages and therefore huge pages are completely isolated and correlation attacks abusing of its low entropy are not longer possible.
- Specific-zone for thread stacks: If enabled, thread stacks are allocated in a designated specific-zone. This no only prevents correlation attacks but also separate its data content from the libraries since both are by default in the same area.
- Inter-Object to Stack, Executable and Heap: each one of these objects is independently randomized, which is the default behavior for Linux and PaX. It was added to support the concentrated mode by disabling it.
- Randomize specific-zones per child: When a new child is spawned, all specific-zones are renewed, which results in a different memory map between the parent and the child, as well as any siblings among them.
- Sub-page in heap and thread stacks: applies sub-page randomization to the thread stacks and the heap. This feature can also be used from user-land on a per object basis, by calling the `mmap()` with the new flag `MAP_INTRA_PAGE`.
- Isolate thread stacks: randomizes thread stacks individually. This feature can also be requested by using the `MAP_RND_OBJECT` flag when calling `mmap()`.

- Isolate LD and vDSO: by enabling this feature, ASLR-NG loads these objects individually instead of using the classic library/mmap zone.
- Bit-slicing: enabling this feature, ASLR-NG generates a random number at boot time which is later used to improve the entropy of some objects when they must be aligned, typically for cache aliasing performance. Instead of setting the sensitive bits to zero, they are set to the random value generated at boot. We have used the core idea of this novel randomization form to address a security issue in the Linux kernel 4.1, to increase entropy by 3 bits in the AMD Bulldozer processor family [31].
- Isolate all objects: all objects are independently randomized. The leakage of any object cannot be used to de-randomize any other. This feature can be used in very exposed or critical environments where security is paramount.

8. Evaluation

This section compares ASLR-NG with Linux, PaX and OS X. Firstly, Section 8.1 compares the main randomization forms to identify the new features introduced by the ASLR-NG. In Section 8.2 the entropy bits for 32 and 64 bits in the x86 architecture are compared, and finally the correlation entropy of the objects is presented.

8.1. Randomization Forms

Linux and PaX provide very few randomization forms, and furthermore they do not generalize them either. For example, they do not provide sub-page or inter-object randomization for thread stacks.

ASLR-NG extends already used forms of entropy to most objects and provides new forms to prevent correlation attacks [27]. It worth mentioning the concept of specific-zones, which is a simple mechanism employed to group together sensitive objects and isolate them from the rest. Table 4 summarizes the main features of Linux, PaX and ASLR-NG.

Table 4. Comparative summary of features.

Feature and Forms	OS X	Linux	PaX	ASLR-NG
ASLR per-exec		✓	✓	✓
Inter-object in stack, exec. and heap	✓	✓	✓	✓
Sub-page in main stack	✓	✓	✓	✓
Sub-page in ARGV and heap (brk)			✓	✓
Inter-object in LD and vDSO				✓
Inter-object in thread stacks				✓
Sub-page in thread stacks				✓
Load libraries order randomized				✓
Multiple specific-zone support				✓
Randomize specific-zones per child				✓
Bit-slicing randomization				✓
Sub-page per mmap request				✓
Inter-object per mmap request				✓
Uniform distribution				✓
Full VM range				✓

8.2. Absolute Address Entropy

Absolute entropy is the effective entropy of an object when it is considered independently. Each ASLR has been tested in two different systems:

- **32-bits:** a 32-bit x86 architecture, without PAE. Note that when an i386 application is executed in a x86_64 system, the memory layout is different. Our experiments are executed in a truly 32-bit system, and so the virtual memory space available to any process is 3 GB.
- **64-bits:** a 64-bit x86_64 architecture. The virtual memory space available for the user is 2^{47} bytes.

Table 5 shows the measured entropy bits obtained in Linux, PaX, OS X and ASLR-NG in both 32- and 64-bit systems. Note that the ASLR-NG absolute entropy is the same for all modes. All the data presented in this section are the result of running the sampler tool to collect a million samples for each system which is more than enough for the virtual memory space. ARGV is the page in memory that hold the program arguments.

Table 5. Comparative summary of bits of entropy.

Object	32-Bits				64-Bits			
	OS X	Linux	PaX	ASLR-NG	OS X	Linux	PaX	ASLR-NG
ARGV	8	11	27	31.5	16	22	39	47
Main stack	8	19	23	27.5	16	30	35	43
Heap (brk)	8.7	13	23.3	27.5	15.9	28	35	43
Heap (mmap)	7.7	8	15.7	27.5	16	28	28.5	43
Thread stacks	11	8	15.7	27.5	16	28	28.5	43
Sub-page object	-	-	-	27.5	-	-	-	43
Regular mmaps	7.7	8	15.7	19.5	16	28	28.5	35
Libraries	7.7	8	15.7	19.5	16	28	28.5	35
vDSO	7.7	8	15.7	19.5	16	21.4	28.5	35
Executable	8	8	15	19.5	16	28	27	35
Huge pages	0	0	5.7	9.5	7	19	19.5	26

Linux: In 32-bit systems, Linux provides only 8 random bits for most objects, which is too low a value to be effective and can be considered defeated. In 64 bits, although randomization is higher for most objects, there are still some objects (vDSO and ARGV) with much lower entropy, which in turn may encourage attackers to use them.

Huge pages are less randomized, due to alignment constraints. In particular, in 32-bit systems, alignment resets those bits that ASLR randomizes, and so huge pages are not randomized at all. Moreover, in 64-bit systems, huge pages have 19 random bits, which gives some protection but still may not deter local or remotely distributed attackers.

PaX/Grsecurity: In 32 bits, PaX provides much more entropy than Linux in all objects. The libraries and mmapped objects have 15.72 bits of entropy, in which case a brute force attack, at 100 trials per second, will need a few minutes to bypass the PaX ASLR. The lowest randomized object (but huge pages) is the executable. Surprisingly in 64-bit systems, the entropy of Executables in PAX is less than the entropy in Linux. The additional entropy bits of the ARGV, main stack and heap are due to sub-page randomization. The decimal values of the mmapped objects are caused by non-uniform distribution—as explained in Section 5.2. PaX is much better than Linux in 32 bits, but quite similar in 64 bits.

OS X: ASLR is broken in OS X for local attackers for both 32- and 64-bit systems and is weak for remote attackers. As Table 4 shows, OS X implements its ASLR per boot. That is, all objects except the executable, are only randomized after the system is rebooted. Therefore it provides no protection against local attackers and its ASLR in both 32 and 64-bits must be considered *broken* for local attackers. The OS X entropy showed in Table 5 only apply for remote attackers. In 32-bit systems it provides a similar protection than Linux but in 64-bit the entropy provided is very low allowing potential attackers to bypass the OS X ASLR with little effort.

Note that because ASLR is applied only when the system is rebooted, remote attackers can do a brute force attack against the libraries without requiring a forking server since the libraries will be always mapped to the same addresses. On average, to remotely bypass the ASLR in 64-bit OS X systems will require $\frac{2^{16}}{2}$ = 32,768 attempts which clearly is not enough to deter attackers.

ASLR-NG: In 32 bits, libraries and mapped objects have almost 20 bits of entropy, which is comparable with the minimum randomized objects in 64-bit Linux (vDSO and ARGV). Because of the small VM space in 32 bits the entropy is intrinsically limited, but thanks to the ability of the ASLR-NG

to use the full address range to allocate any object, it increases entropy by up to 20 more bits than Linux and 12 more than PaX. Although ASLR-NG provides the highest randomization for huge pages, the alignment constraint (which resets the lowest 22 bits) only leaves the possibility of randomizing the highest 10 bits.

In 64 bits, ASLR-NG provides up to 15 more bits than Linux and 14 more than PaX. Regarding huge pages, Linux and PaX have 1 million possible places to load huge pages compared with the 67 million of ASLR-NG. This increment in entropy, jointly with the specific-zone for huge pages, increases the cost for an attacker to guess where they are placed and at the same time prevents using them in correlation attacks. Hence, ASLR-NG outperforms Linux and PaX ASLR in both 32- and 64-bit systems.

8.3. Correlation in ASLR-NG

ASLR-NG addresses correlation weakness by randomizing objects and specific-zones independently. Obviously, all the objects allocated in the same specific-zone are correlated together, but they are uncorrelated in relation to other specific-zones or objects.

The concentrated mode, by definition, is fully correlated to provide a compact layout to systems with low resources. The conservative mode is close to Linux and PaX but prevents using the stack, executable and heap in correlation attacks.

In extended mode, ASLR-NG extends the conservative mode by five specific-zones to isolate objects of different criticality levels. The paranoid mode goes a step beyond, though, by removing the correlation between all pairs of objects (no specific-zones are created), but as far as we know, exploiting the correlation between objects in the same category is not useful. Typically, a single library contains enough gadgets to build ROP exploits, and so it is not necessary to de-randomize other libraries.

9. Conclusions

In this paper we have analyzed the major operating system ASLR implementations to assess its effectiveness and its weakness from the local and remote attackers point of view, including the impact in the IoT devices based in Linux and OS X.

To do the assessment, we have proposed a taxonomy of all ASLR elements creating a categorization of three entropy dimensions. Based on this taxonomy we have created ASLRA, an advanced statistical analysis tool to automatically assess the effectiveness of any ASLR implementation.

Our analysis showed that all ASLR implementations suffer from several weaknesses, 32-bit systems provide a poor ASLR, and OS X has a broken ASLR in both 32- and 64-bit systems. This is jeopardizing not only servers and end users devices as smartphones but also the growing IoT ecosystem. We have then presented ASLR-NG, a novel ASLR that provides the maximum possible absolute entropy and removes all correlation attacks making ASLR-NG the best solution for both 32- and 64-bit systems. We implemented ASLR-NG in the Linux kernel 4.15 showing that ASLR-NG overcomes PaX, Linux and OS X implementations, providing strong protection to prevent attackers from abusing weak ASLRs.

Author Contributions: Writing—Original draft, H.M.-G. and I.R.R.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Aga, M.T.; Austin, T. Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization. In Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Washington, DC, USA, 16–20 February 2019; pp. 26–36, doi:10.1109/CGO.2019.8661202. [CrossRef]
2. Jelinek, J. Object Size Checking to Prevent (Some) Buffer Overflows (GCC FORTIFY). 2004. Available online: http://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html (accessed on 17 July 2019).

3. Shahriar, H.; Zulkernine, M. Mitigating Program Security Vulnerabilities: Approaches and Challenges. *ACM Comput. Surv.* **2012**, *44*, 11, doi:10.1145/2187671.2187673. [CrossRef]
4. Carlier, M.; Steenhaut, K.; Braeken, A. Symmetric-Key-Based Security for Multicast Communication in Wireless Sensor Networks. *Computers* **2019**, *8*, 27, doi:10.3390/computers8010027. [CrossRef]
5. Choudhary, J.; Balasubramanian, P.; Varghese, D.M.; Singh, D.P.; Maskell, D. Generalized Majority Voter Design Method for N-Modular Redundant Systems Used in Mission- and Safety-Critical Applications. *Computers* **2019**, *8*, 10, doi:10.3390/computers8010010. [CrossRef]
6. Shacham, H.; Page, M.; Pfaff, B.; Goh, E.J.; Modadugu, N.; Boneh, D. On the effectiveness of address-space randomization. In Proceedings of the 11th ACM Conference on Computer and Communications Security, Washington, DC, USA, 25–29 October 2004; ACM: New York, NY, USA, 2004; pp. 298–307, doi:10.1145/1030083.1030124. [CrossRef]
7. Marco-Gisbert, H.; Ripoll, I. Preventing Brute Force Attacks Against Stack Canary Protection on Networking Servers. In Proceedings of the 12th International Symposium on Network Computing and Applications, Cambridge, MA, USA, 22–24 August 2013; pp. 243–250, doi:10.1109/NCA.2013.12. [CrossRef]
8. Marco-Gisbert, H.; Ripoll, I. On the effectiveness of NX, SSP, RenewSSP and ASLR against stack buffer overflows. In Proceedings of the 13th International Symposium on Network Computing and Applications, Cambridge, MA, USA, 21–23 August 2014; pp. 145–152.
9. Friginal, J.; de Andrés, D.; Ruiz, J.C.; Gil, P.J. Attack Injection to Support the Evaluation of Ad Hoc Networks. In Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems, New Delhi, India, 31 October–3 November 2010; pp. 21–29, doi:10.1109/SRDS.2010.11. [CrossRef]
10. Xu, J.; Kalbarczyk, Z.; Iyer, R. Transparent runtime randomization for security. In Proceedings of the 22nd International Symposium on Reliable Distributed Systems, Florence, Italy, 6–8 October 2003; pp. 260–269, doi:10.1109/RELDIS.2003.1238076. [CrossRef]
11. Zhan, X.; Zheng, T.; Gao, S. Defending ROP Attacks Using Basic Block Level Randomization. In Proceedings of the 2014 IEEE Eighth International Conference on Software Security and Reliability-Companion, San Francisco, CA, USA, 30 June–2 July 2014; pp. 107–112, doi:10.1109/SERE-C.2014.28. [CrossRef]
12. Kil, C.; Jim, J.; Bookholt, C.; Xu, J.; Ning, P. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In Proceedings of the 2006 22nd Annual Computer Security Applications Conference (ACSAC'06), Miami Beach, FL, USA, 11–15 December 2006; pp. 339–348.
13. Iyer, V.; Kanitkar, A.; Dasgupta, P.; Srinivasan, R. Preventing Overflow Attacks by Memory Randomization. In Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering, San Jose, CA, USA, 1–4 November 2010; pp. 339–347, doi:10.1109/ISSRE.2010.22. [CrossRef]
14. Van der Veen, V.; dutt Sharma, N.; Cavallaro, L.; Bos, H. Memory Errors: The Past, the Present, and the Future. In Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses, Amsterdam, The Netherlands, 12–14 September 2012; Springer: Berlin/Heidelberg, Germany, 2012; pp. 86–106, doi:10.1007/978-3-642-33338-5_5. [CrossRef]
15. Pax Team. PaX Address Space Layout Randomization (ASLR). 2003. Available online: http://pax.grsecurity.net/docs/aslr.txt (accessed on 17 July 2019).
16. Kocher, P.; Horn, J.; Fogh, A.; Genkin, D.; Gruss, D.; Haas, W.; Hamburg, M.; Lipp, M.; Mangard, S.; Prescher, T.; et al. Spectre Attacks: Exploiting Speculative Execution. *arXiv* **2019**, arXiv:1801.01203.
17. Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Fogh, A.; Horn, J.; Mangard, S.; Kocher, P.; Genkin, D.; et al. Meltdown: Reading Kernel Memory from User Space. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), San Francisco, CA, USA, 16–20 April 2018.
18. Edge, J. Kernel Address Space Layout Randomization. 2013. Available online: https://lwn.net/Articles/569635 (accessed on 17 July 2019).
19. Rahman, M.A.; Asyhari, A.T. The Emergence of Internet of Things (IoT): Connecting Anything, Anywhere. *Computers* **2019**, *8*, 40, doi:10.3390/computers8020040. [CrossRef]
20. Bojinov, H.; Boneh, D.; Cannings, R.; Malchev, I. Address space randomization for mobile devices. In Proceedings of the Fourth ACM Conference on Wireless Network Security, Hamburg, Germany, 14–17 June 2011; ACM: New York, NY, USA, 2011; pp. 127–138, doi:10.1145/1998412.1998434. [CrossRef]
21. Lu, K.; Nürnberger, S.; Backes, M.; Lee, W. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In Proceedings of the 23rd Annual Symposium on Network and Distributed System Security (NDSS 2016), San Diego, CA, USA, 21 February 2015.

22. Hiser, J.; Nguyen-Tuong, A.; Co, M.; Hall, M.; Davidson, J. ILR: Where'd My Gadgets Go? In Proceedings of the 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 571–585, doi:10.1109/SP.2012.39.
23. Xu, H.; Chapin, S.J. Address-space layout randomization using code islands. *J. Comput. Secur.* **2009**, *17*, 331–362. [CrossRef]
24. Crane, S.; Liebchen, C.; Homescu, A.; Davi, L.; Larsen, P.; Sadeghi, A.R.; Brunthaler, S.; Franz, M. Readactor: Practical code randomization resilient to memory disclosure. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; pp. 763–780.
25. Wartell, R.; Mohan, V.; Hamlen, K.W.; Lin, Z. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In Proceedings of the 2012 ACM Conference on Computer and Communications Security, Raleigh, NC, USA, 16–18 October 2012; ACM: New York, NY, USA, 2012; pp. 157–168, doi:10.1145/2382196.2382216. [CrossRef]
26. Lin, Z.; Riley, R.D.; Xu, D. Polymorphing Software by Randomizing Data Structure Layout. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*; Flegel, U., Bruschi, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 107–126.
27. Marco-Gisbert, H.; Ripoll, I. On the Effectiveness of Full-ASLR on 64-bit Linux. In Proceedings of the In-Depth Security Conference 2014 (DeepSec), Vienna, Austria, 18–21 November 2014.
28. Bittau, A.; Belay, A.; Mashtizadeh, A.; Mazières, D.; Boneh, D. Hacking Blind. In Proceedings of the 35th IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–21 May 2014. Available online: http://www.ieee-security.org/TC/SP2014/papers/HackingBlind.pdf (accessed on 17 July 2019).
29. Drepper, U. Growable Maps Removal. 2008. Available online: https://lwn.net/Articles/294001/ (accessed on 17 July 2019).
30. Lefevre, V. Silent Stack-Heap Collision under GNU/Linux. 2014. Available online: https://gcc.gnu.org/ml/gcc-help/2014-07/msg00076.html (accessed on 17 July 2019).
31. Marco-Gisbert, H.; Ripoll, I. AMD Bulldozer Linux ASLR Weakness: Reducing Entropy by 87.5%. 2015. Available online: http://hmarco.org/bugs/AMD-Bulldozer-linux-ASLR-weakness-reducing-mmaped-files-by-eight.html (accessed on 17 July 2019).
32. Marco-Gisbert, H.; Ripoll, I. CVE-2015-1593—Linux ASLR Integer Overflow: Reducing Stack Entropy by Four. 2015. Available online: http://hmarco.org/bugs/linux-ASLR-integer-overflow.html (accessed on 17 July 2019).
33. Marco-Gisbert, H.; Ripoll, I. Linux ASLR Mmap Weakness: Reducing Entropy by Half. 2015. Available online: http://hmarco.org/bugs/linux-ASLR-reducing-mmap-by-half.html (accessed on 17 July 2019).
34. Beirlant, J.; Dudewicz, E.J.; Györfi, L.; Van der Meulen, E.C. Nonparametric entropy estimation: An overview. *Int. J. Math. Stat. Sci.* **1997**, *6*, 17–39.
35. Lesne, A. Shannon entropy: A rigorous notion at the crossroads between probability, information theory, dynamical systems and statistical physics. *Math. Struct. Comput. Sci.* **2014**, *24*, e240311, doi:10.1017/S0960129512000783. [CrossRef]
36. Kozachenko, L.F.; Leonenko, N.N. Sample estimate of the entropy of a random vector. *Probl. Inf. Transm.* **1987**, *23*, 95–101.
37. 'pi3' Zabrocki, A. Scraps of Notes on Remote Stack Overflow Exploitation. 2010. Available online: http://www.phrack.org/issues.html?issue=67&id=13#article (accessed on 17 July 2019).
38. Herlands, W.; Hobson, T.; Donovan, P.J. Effective entropy: Security-centric metric for memory randomization techniques. In Proceedings of the 7th Workshop on Cyber Security Experimentation and Test (CSET 14), San Diego, CA, USA, 2014.
39. Uchenick, G.M.; Vanfleet, W.M. Multiple independent levels of safety and security: High assurance architecture for MSLS/MLS. In Proceedings of the MILCOM 2005—2005 IEEE Military Communications Conference, Atlantic City, NJ, USA, 17–20 October 2005; Volume 1, pp. 610–614, doi:10.1109/MILCOM.2005.1605749. [CrossRef]
40. Rohlf, C.; Ivnitskiy, Y. *Attacking Clientside JIT Compilers*; Black Hat: Las Vegas, NV, USA 2011.
41. Lee, B.; Lu, L.; Wang, T.; Kim, T.; Lee, W. From Zygote to Morula: Fortifying Weakened ASLR on Android. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 18–21 May 2014; pp. 424–439, doi:10.1109/SP.2014.34. [CrossRef]

42. The Heartbleed Bug. 2014. Available online: http://heartbleed.com (accessed on 17 July 2019).
43. Wilson, P.; Johnstone, M.; Neely, M.; Boles, D. Dynamic storage allocation: A survey and critical review. In *Memory Management*; Baler, H., Ed.; Springer: Berlin/Heidelberg, Germany, 1995; Volume 986, pp. 1–116, doi:10.1007/3-540-60368-9_19.